New Directions in Software Quality Assurance Automation

Mikhail Auguston

Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
Phone: (831)-656-2607
Email: maugusto@nps.edu

Abstract. A formalism is suggested for specifying environment behavior models for software test scenario generation based on attributed event grammars. The environment model may contain descriptions of the events triggered by the software outputs and of the hazardous states in which the system could arrive, thus providing a framework for specifying properties of software behavior within the given environment. The behavior of the system can be rendered as an event set with two partial ordering relations: precedence and inclusion (event trace). This formalism may be used as a basis for automation tools for test generation, test result monitoring and verification, for experiments to gather statistics about software safety, and for evaluating of dependencies of system's behavior on environment parameters. The monitoring activities can be implemented within a uniform framework as computations over event traces.

Keywords: environment models, reactive systems, requirements specification and verification, testing and safety assessment automation, event traces.

1 Introduction

Reactive and real-time systems are at the core of many safety-critical software applications. In [1][2][3][4] an approach to testing automation for reactive and real-time software systems based on attributed event grammars (AEG) has been introduced. The main idea is to specify the environment behavior model as a set of events that control the inputs for the system under the test (SUT) and that may adjust the behavior depending on the outputs provided by the SUT (adaptive testing [14]).

2. The Environment Model

The notion of event is central for our approach. An **event** is any detectable action in the environment that could be relevant to the operation of the SUT. A keyboard button pressed by the user, a group of alarm sensors triggered by an intruder, a particular stage of a chemical reaction monitored by the system, and the detection of an enemy missile are examples of events. In our approach

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **JUN 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **New Directions in Software Quality Assurance Automation**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Computer Science,1 University Circle,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

In Proceedings of the 14th International Command and Control Research and Technology Symposium (ICCRTS) was held Jun 15-17, 2009, in Washington, DC

14. ABSTRACT

see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**64**	

an event usually is a time interval, and has a beginning, an end, and duration. An event has **attributes**, such as type and timing attributes.

There are two basic relations defined for events: **precedence** (PRECEDES) and **inclusion** (IN). Two events may be ordered, or one event may appear inside another event. The behavior of the environment can be represented as a set of events with these two basic relations defined for them (**event trace**). Usually event traces have a certain structure (or constraints) in a given environment. The basic relations define two partial orders of events. For example, two events are not necessarily ordered under the PRECEDES relation, that is, they can happen concurrently.

The structure of possible event traces can be specified by **event grammar**. Here identifiers stand for event types, sequence denotes precedence of events, (…|…) denotes alternative, * means repetition zero or more times of ordered events, {a, b} denotes a set of two events a and b without an ordering relation between them, and {…}* denotes a set of zero or more events without an ordering relation between them. The rule *A::= B C* means that an event of the type *A* contains (IN relation) ordered events of types *B* and *C* correspondingly (PRECEDES relation).

Example 1

OfficeAlarmSystem::= {DoorMonitoring,
WindowMonitoring }

The *OfficeAlarmSystem* run is a set of two concurrent monitoring threads.

DoorMonitoring::= DoorSensor*

The *DoorMonitoring* is a composite event, which contains a sequence of ordered events of the type *DoorSensor.*

WindowMonitoring::= WindowSensor*

DoorSensor::= (DoorClosed | DoorAlarm)

The *DoorSensor* event may contain one of two possible alternatives.

WindowSensor::= (WindowClosed | WindowAlarm)

This event grammar defines a set of possible event traces – a model of a certain environment. The purpose is to use it as a production grammar for random event trace generation by traversing grammar rules and making random selections of alternatives and numbers of repetitions.

2.1 Event Attributes

An event may have attributes and actions associated with it. Each event type may have a different attribute set. Event grammar rules can be decorated with attribute evaluation rules. The */action/* is performed immediately after the preceding event is completed. Events usually have timing attributes like *begin_time*, *end_time*, and *duration*. Some of those attributes can be defined in the grammar by appropriate actions, while others may be calculated by appropriate default rules. Attributes can be either inherited or synthesized, **we assume that all attribute evaluations are accomplished in a single pass and the event grammar is traversed top-down, left-to-right for producing a particular event trace.** The interface with the SUT can be specified by an action that sends input values to the SUT or listens for a message sent by the SUT. This may be a subroutine in a common programming language like C or Java that hides the necessary wrapping code.

Example 2.

An (over)simplified environment model for a missile defense system that tracks radar sensors and at certain moment sends a command to proceed with an interception.

Attack::= { Missile_launch } * (=N)

The *Attack* event contains N parallel *Missile_launch* events.

```
Missile_launch::=
    Boost_stage
    Middle_stage
    WHEN(Middle_stage.completed)    Boom
```

The *Boom* event (which happens if the interception attempts have failed) represents an environment event, which the SUT should try to avoid, or a "hazard state" in which the system may arrive.

```
Middle_stage::=
   / Middle_stage.completed := True/
    (    move
         CATCH  SUT_launch_interception(hit_coordinates)
         WHEN (hit_coordinates == Middle_stage.coordinates )
```

```
            [ p(p1) interception
                      / Middle_stage.completed := False;
                       send_hit_input( Middle_stage .coordinates);
                        BREAK; /   ]
  ) * (<=M, EVERY 50 msec)
```

The sequence of *move* events within *Middle_stage* event may be interrupted by receiving an external event from the SUT. This will suspend the *move* event sequence and will either continue with event *interception* (with probability p1), which simulates the missile interception event triggered by the SUT, followed by the *BREAK* command, which terminates the event iteration, or will resume the *move* sequence. This model allows several interception attempts during the same *Middle_stage* event. In general, external events generated by the SUT may be broadcasted to several event listeners in the AEG, or may be exclusive and be consumed by just one of the listeners. These interface details are encapsulated in the listener Boolean subroutines like *SUT_launch_interception(hit_coordinates)* where the parameter *hit_coordinates* is passed by reference.

```
move ::=  /        adjust( ENCLOSING Middle_stage .coordinates) ;
                    send_radar_signal(ENCLOSING Middle_stage.coordinates);  /
```

This rule provides attribute calculations and sends an input to the SUT simulating the inputs from radar sensors. The *ENCLOSING* construct provides access to the attributes of parent event.

It should be pointed out that most of the event trace generation and attribute evaluation can be accomplished during the generation time, and the test driver extracted from the event trace contains only actions and their time stamps (like send/catch subroutine calls) that should be postponed to the run time. This makes it amenable for fulfilling real time constraints for the input streams needed to be fed into SUT. The event trace provides a “scaffold” for building a light-weight and efficient test driver. Since the event trace generation from the AEG still may contain random elements, like alternative and number of iteration selection, the number of different scenarios generated from the same AEG is potentially unlimited.

3. Behavior Properties Specification

The next problem to be addressed after the system behavior model is set up is the formalism specifying properties of the behavior. As a unifying framework we came up with the concept of a computation over the event trace. This approach implies the design of a special programming language for computations over the event traces. In [6], [8], [7], [9] a language FORMAN, based on functional paradigm and the use of event patterns and aggregate operations over events, is suggested.

Event patterns describe the structure of events with possible context conditions. Execution paths can be described by path expressions over events. This makes it possible to write assertions not only about pre-conditions and post-conditions at event trace points, but also about data flows in the entire trace.

The subroutine calls for inputs in the SUT and for catching outputs from the SUT can be considered also as events with obvious precedence and inclusion relations with the rest of event trace. The parameter values at the beginning and the end of those events are specific attributes that provide the opportunity to write assertions about system input/output values at different points in the execution history.

3.1 The Language for Computations over Event Traces

FORMAN is a high-level specification language for expressing intended behavior or known types of error conditions when debugging or testing programs. FORMAN supplies a means for writing assertions about events and event sequences and sets. Monitoring activities can be implemented as computations over event traces. Typical examples of monitoring include:

- Assertion checking (test oracles)
- Debugging queries
- Profiles
- Performance measurements
- Behaviour visualization

The following provides an outline of the FORMAN constructs. More details are available in [6][8][7]. The environment model from Example 2 will be used as a background for further examples.

Event patterns

```
x: Middle_stage & x.Value_at_end(completed)== False
```

This pattern matches an event of the type `Middle_stage` if and only if the value of the `completed` attribute at the end of this event is `False`.

List of events

Assuming that `m` is an event of the type `Middle_stage.`

```
[ move FROM m ]
```

This creates a list of `move` events from the enclosing even m preserving the precedence relation between them.

List of values

Assuming that `m` is an event of the type `Middle_stage.`

```
[ x: move FROM m APPLY x.Value_at_end( m.coordinates
) ]
```

This creates a list of values of `coordinates` attribute of the enclosing `Middle_stage` event `m` taken at the end of each `move` event inside `m`. Note that the value of `m.coordinates` may change after each `move` event.

Aggregate operations

Assuming that `m` is an event of the type `Middle_stage.`

```
OR/[ x: SUT_launch_interception FROM m

    APPLY x.param[1] == x.Value_at_end( m.coordinates )]
```

This expression yields a Boolean value depending on whether there is at least one instance x of `SUT_launch_interception` event inside m that yields True for the expression `x.param[1] == x.Value_at_end(m.coordinates).` The `x.param[1]` denotes the value of the first actual parameter of the subroutine `SUT_launch_interception` call. This aggregate operation can be abbreviated as:

```
EXISTS x: SUT_launch_interception FROM m

    ( x.param[1] == x.Value_at_end( m.coordinates ))
```

In a similar way, FOREACH quantifier can be introduced as an abbreviation for the AND/ aggregate operation.

Generic requirements for the SUT behaviour within the given environment can be specified in FORMAN. The following examples illustrate this.

Example 3.

The requirements for the SUT may include for example the following: “There is at least one interception attempt for each `Missile_launch` event within the `Attack` event.”

```
FOREACH x: Missile_launch FROM Attack

     EXISTS y: SUT_launch_interception FROM x
```

Example 4.

The first interception attempt should happen no later than 1 sec after the beginning of the `Missile_launch` event.

```
FOREACH x: Missile_launch FROM Attack

     EXISTS y: SUT_launch_interception FROM x

          y.begin_time – x.begin_time < 1 sec
```

Example 5.

There should not be unintercepted missile launches.

```
CARD/[ Boom FROM Attack]  == 0
```

The examples of FORMAN expressions above represent computations over the event traces and can be performed during the test run or after it based on a log file collected during the test run. This supports the requirement tracing as a part of testing process.

This framework provides means for expressing quantifiers over events and ordering and inclusion relations for events and is comparable with the expressive power of other specification formalisms for behavior specification, such as temporal logic and abstract event traces [15], [16].

Figure 1 outlines the testing automation architecture based on AEG.

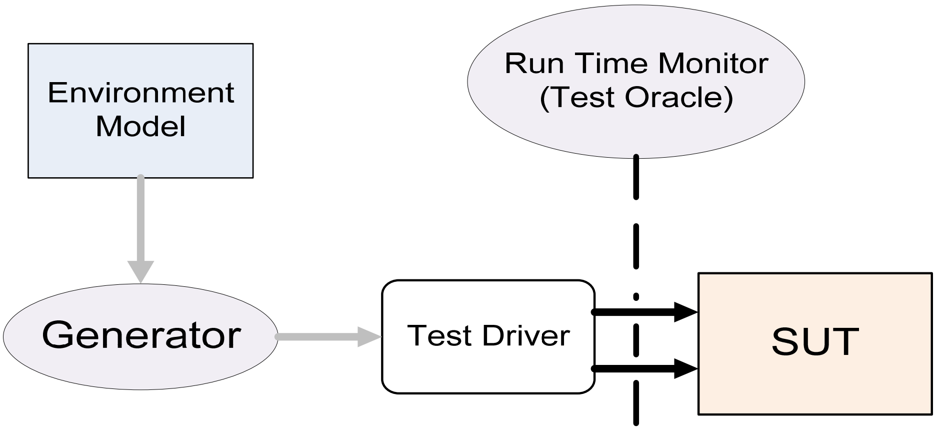


Figure 1. The use of the automated test generator.

4. Automated Safety Assessment

NASA-STD-8719.13A [24] defines **risk** as a function of the possible frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity. It may be a challenge to estimate those for real systems because of size and lack of good analytical model. Here we suggest a pragmatic approach to this problem. An environment model may contain events and attributes representing some hazard situations that may occur during the run time as a result of SUT interaction with the environment. This feature of the AEG model provides a basis for automated system safety analysis. We can estimate the risk of arriving in a hazard state by running scenarios of SUT interacting with the environment model.

In the previous example, the *Boom* event occurs in certain scenarios depending on the SUT outputs received by the test driver and random choices determined by the given probabilities. From the point of view of SUT this is a highly undesirable event. If we run a large enough number of (automatically generated) tests, the statistics gathered give some approximation for the risk of getting to this hazardous state. This becomes a simple constructive process of performing experiments with SUT behavior within the given environment model (***"software-in-the-loop"*** simulations). Large sets of different scenarios (and, respectively, test cases extracted from them) can be generated from the same AEG model since each scenario generation is based on some (pseudo)random choices during the generation process.

4.1 Parameterized Safety Analysis

We can do a qualitative analysis as well and ask questions like “what has contributed to this outcome?” We can change some parameters of the environment model, or change some parameters in the SUT and repeat the set of tests. If the frequency of reaching a hazardous state changes, we can answer the question asked. These kinds of experiments with model parameters could be done automatically in a systematic way.

Experimenting with increasing or decreasing the number of missile launches *N*, the duration of particular missile launch M, and the probability of interception *p1* in the previous example, we can determine what impact those parameters have on the probability of hazardous outcome, and find thresholds for SUT behavior in terms of *N, M,* and *p1* values.

We suggest to use the ***combinatorial testing technique*** based on orthogonal arrays [19], an approach well familiar to statisticians, to conduct the experiments with parameterized environment models. In 1997, researchers at Telcordia Technologies (formerly Bell Communications Research, or Bellcore) published a paper by Siddharta Dalal et al., “The Combinatorial Design Approach to Automatic Test Generation [18].” Telcordia's studies suggest that “most field faults were caused by either incorrect single values or by an interaction of pairs of values.” If that's generally correct, we ought to focus our testing on the risk of single-mode and double-mode faults. The same conjecture that stipulates that the fault in behavior of the SUT in most cases depends either on a single parameter value or on an interaction of a pair of parameter values could be applied to the system safety testing.

The rationale for using orthogonal arrays for experiments with the SUT is similar to the rationale for the use of orthogonal arrays for experiments in other engineering domains [20], [22], [23]. The use of an orthogonal array guarantees that all pair-wise samples are represented evenly for statistical purposes.

Combinatorial approach will significantly reduce the number of experiments needed to establish statistically sound conclusions about probabilities to reach hazard states for different environment model settings. In order to apply combinatorial testing techniques the values of model parameters have to be split into a finite number of equivalence classes, a technique well known in software component testing [21].

Figure 2 outlines the major steps in the testing and safety assessment process based on AEG.

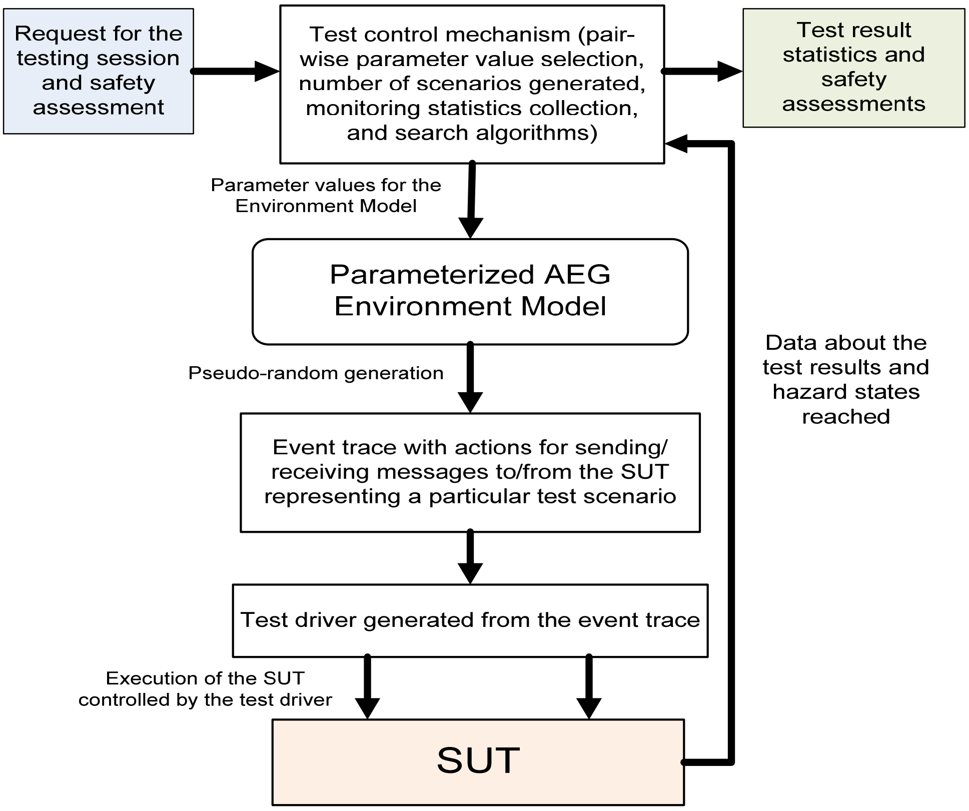

Request for the testing session and safety assessment
Test control mechanism (pair-wise parameter value selection, number of scenarios generated, monitoring statistics collection, and search algorithms)
Test result statistics and safety assessments
Parameter values for the Environment Model
Parameterized AEG Environment Model
Data about the test results and hazard states reached
Pseudo-random generation
Event trace with actions for sending/ receiving messages to/from the SUT representing a particular test scenario
Test driver generated from the event trace
Execution of the SUT controlled by the test driver
SUT

Figure 2. Testing and system safety assessment automation framework based on attributed event grammars as environment models.

5. Related work

Traditionally, modeling approaches used for software development focus on the system under development. These models emphasize the reactive aspects of the system behavior, which are typically modeled using statechart formalism. In contrast, the purpose of the environment model is to generate stimuli for the system under test. An environment model emphasizes the productive aspects of the behavior.

It has become a common practice for engineers to analyze system behaviors from an external point of view using use cases. In UML (Unified Modeling Language) [26] use case scenarios are written in natural language and focus on the events and responses between the actors and the system. Functional requirements can be derived from the description of events received by the system and the expected responses generated by the system.

The major paradigms for modeling system behavior are based on different variations of finite state machines. Active research in this area focuses on different aspects of behavior specification based on UML statecharts, message sequence diagrams, or other types of extended finite state machines, like timing automata [27] or Petri nets.

State machines are typically used for modeling systems. System models are built around the notion of a transition in response to the environment stimulus. Grammars are common vehicles for generating structured sets of inputs. While grammars and state machines are considered to be dual, researchers have long recognized the power of state machines as acceptors and grammars as generators.

A major feature of our approach is the notion of an event trace as a formal model of behavior. Event grammars are one of the possible frameworks to utilize this notion. They are text-based, have a smaller semantic distance from the use case scenarios than the state machines, and are well suited to model environments described via use case scenarios. Event grammars are convenient in specifying dynamic environments with an arbitrary number of actors (and concurrent events), whereas state machines are effective for modeling static environments (with a predetermined numbers of actors).

In [28], Wang and Parnas proposed to use trace assertions to formally specify the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior of the module specified. Their approach is based on algebraic specifications and term rewriting techniques and is only applicable to non-real-time applications.

In [29], Alfonso et al. presented a formal visual language for expressing real-time system constraints as event scenarios (events and responses) and a tool to translate the scenarios into observer timed automata, which can be used to study properties of the formal model of the system under analysis via model checking and run-time verification. While there are a lot of similarities between the approach presented in [29] and ours, the former is effective for modeling static environments (with fixed scenarios) whereas ours, which is based on event grammar, is more effective in specifying dynamic environments with an arbitrary number of actors (and concurrent events).

Context-free grammars have been used for test generation, in particular, to check compiler implementation, such as in [30] and [31]. Maurer’s article [31] provides an outlook in the use of enhanced context-free grammars for generation of test data.

6. Advantages of the suggested approach

Test result verification is an important aspect of testing automation. The AEG approach assumes that all interaction between the SUT and environment model flows through the subroutine calls attached to the environment events. This implies that it will be straightforward to instrument the interface points with necessary code to monitor and verify the information flow between the SUT and the environment model. In fact, this template closely resembles the Aspect-Oriented Programming paradigm [33].

Traditionally reactive systems and their environments are modeled with some kind of finite state machine, like statecharts or timing automata. For the purposes of scenario (and corresponding test case) generation, the AEG approach may have several useful features, in particular:

- It is based on a precise behavior model in terms of an event trace with precedence and inclusion relations, well suited to capture hierarchical and concurrent behaviors. Since an event may be shared by other events, the model can represent synchronization events as well.
- The control structure suggested by the event grammar notation (sequence, alternative, iteration, concurrent event set) and the top-down, left-to-right order of traversal seems to be intuitive and close to the traditional imperative programming style, hence facilitating the design of models.
- Data flow of attributes is integrated with the control flow (i.e., event trace), and AEG notation provides means for ease of navigation within the derivation tree (e.g., the ENCLOSING event construct for referencing parent event attributes on any distance in the derivation tree).
- The probabilities for alternatives or number of iterations may be attached to meaningful events in the model and are more intuitive and less numerous than in Markov models based on finite state machines. This provides for a natural definition of *functional profiles* for scenario generation.

The main advantages of the suggested approach may be summarized as follows.

- Environment models specified by attributed event grammars provide for automated generation of a large number of pseudo-random (but satisfying the constraints) test drivers. This feature provides for gathering of large enough statistical data for safety assessment experiments.
- All attribute values which don't depend on the SUT output can be calculated at the generation time. As a result the generated test driver contains only actions that should be postponed to the run time (like sending inputs to the SUT and listening to the SUT outputs), has a low overhead, and could be used as a real-time test driver.
- As any notation based on formal grammars AEG is well structured, hierarchical, and scalable.
- The environment model may contain events which represent hazardous states of the environment. Experiments with the SUT embedded in the environment model ("software-in-the-loop") provide a constructive method for quantitative and qualitative assessment of software safety.
- Different environment models for different purposes can be designed, such as for testing extreme scenarios by increasing probability or number of certain events, or for load testing. The same safety assessment methodology as described above may be applied for these special cases as well.
- The environment model itself is an asset and could be reused.
- It addresses the regression testing problem – generated test drivers can be saved and reused. We expect that environment models will be changed relatively seldom unless serious requirement errors are discovered during testing.
- Event traces generated from the AEG model represent examples of SUT interaction with the environment, and are in fact use cases, that could be useful for requirements specification and other prototyping tasks.

The novelty of our approach is in the notion of a formal system behavior model based on event grammars for automated generation of test scenarios and test drivers.

C2 systems to a large degree are reactive and real-time systems, and therefore can benefit from the AEG approach. Our previous work has provided a basis for testing and debugging automation tool design within this framework **Error! Reference source not found.**[9][10][11][12][13]. The feasibility has been proven by the first prototype implementation of AEG **Error! Reference source not found.**[2][3] and case studies, like Infusion

Pump example **Error! Reference source not found.** and environment models for US Marine Corps Technology Center.

References

[1] M.Auguston, B.Michael, M.Shing, "Environment Behavior Models for Automation of Testing and Assessment of System Safety," Information and Software Technology, Elsevier, Volume 48, Issue 10, October 2006, pp. 971-980

[2] Mikhail Auguston, James Bret Michael, Man-Tak Shing, Environment Behavior Models for Scenario Generation and Testing Automation, in Proceedings of the First International Workshop on Advances in Model-Based Software Testing (A-MOST'05), the 27th International Conference on Software Engineering ICSE'05, May 15-16, 2005, St. Louis, USA, http://a-most.argreenhouse.com , This article has also been published on-line in the ACM SIGSOFT Software Engineering Notes Volume 30 , Issue 4 (July 2005).

[3] Mikhail Auguston, James Bret Michael, Man-Tak Shing, Test Automation and Safety Assessment in Rapid Systems Prototyping, in Proceedings of 16th IEEE International Workshop on Rapid System Prototyping, June 8-10, 2005, Montreal, Canada, pp.188-194.

[4] Mikhail Auguston, James Bret Michael, Man-Tak Shing, and David L. Floodeen, "Using Attributed Event Grammar Environment Models for Automated Test Generation and Software Risk Assessment of System-of-Systems", in the Proceedings of 2005 IEEE International Conference on Systems, Man, and Cybernetics, Special Session on Recent Advances in Engineering Systems-of-Systems to Support Joint and Coalition Warfighters, October 10-12, 2005, The Big Island, Hawaii, USA, pp.1870-1875

[5] Harsha Tummala, James Bret Michael, Man-Tak Shing, Mikhail Auguston, David Little, Zachary Pace, Implementation and Analysis of Environment Behavior Models as a Tool for Testing Real-Time, Reactive System, in Proceedings of the 2006 IEEE International Conference on System of Systems Engineering, Los Angeles, CA, USA - April 2006. pp. 260-265

[6] M. Auguston, "FORMAN -- A Program Formal Annotation Language," Proceedings of the 5th Israel Conference on Computer Systems and Software Engineering, Gerclia, May 1991, IEEE Computer Society Press, pp.149-154.

[7] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging," The Journal of Systems and Software 25, 1994, pp. 223-239.

[8] M. Auguston, "Program Behavior Model Based on Event Grammar and Its Application for Debugging Automation," in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995.

[9] M. Auguston, "Building Program Behavior Models," in Proceedings of European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England, August 23-28, 1998, pp. 19-26.

[10] M.Auguston, Assertion Checker for the C Programming Language based on Computations over event traces, in Proceedings of the Fourth International Workshop on Algorithmic and Automatic Debugging, AADEBUG'2000, Munich, Germany, August 28-30, 2000, pp.90-99 on-line proceedings at http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html .

[11] M.Auguston, C.Jeffery, S.Underwood, A Framework for Automatic Debugging, in Proceedings of the 17th IEEE International Conference on Automated Software Engineering, September 23-27, 2002, Edinburgh, UK, IEEE Computer Society Press, pp.217-222.

[12] Mikhail Auguston, Clinton Jeffery, Scott Underwood, A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization, in the Proceedings of 5th International Workshop on Algorithmic and Automatic Debugging AADEBUG 2003, Ghent, Belgium, September 8-10, 2003, pp. 41-54 (also on the CoRR web site at http://arxiv.org/abs/cs/0310025) .

[13] C.Jeffery, M.Auguston, S.Underwood, Towards Fully Automatic Execution Monitoring, in Proceedings of Radical Innovations of Software and Systems Engineering in the Future: 9th International Monterey Workshop, RISSEF 2002, Venice, Italy, Oct. 2002, Revised Papers (Editors: Martin Wirsing, Alexander Knapp, Simonetta Balsamo), Lecture Notes in Computer Science, Springer Verlag, Vol. 2941, March 2004, pp. 204 – 218.

[14] R. M.Hierons, H.Ural, "Concerning the Ordering of Adaptive Test Sequences," in Proc. 23rd IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems, Berlin, Germany, Sept. 2003, Berlin: Springer, Lecture Notes in Computer Science, Vol. 2767, pp. 289-302.

[15] Zohar Manna and Amir Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification,. Springer-Verlag, 1992.

[16] A.Mazurkiewicz, Trace Theory, in W. Brauer et al., editors, Petri Nets, Applications and Relationship to other Models of Concurrency, Vol. 255, Lecture Notes in Computer Science, Springer Verlag, 1987, pp.279-324.

[17] IBM Combinatorial Test Services, http://www.alphaworks.ibm.com/tech/cts

[18] Cohen D. M., Dalal S. R., Fredman M. L., and Patton G. C., *The AETG System: An approach to Testing Based on Combinatorial Design.* IEEE Transactions on Software Engineering, 23 (1997), pp. 437-444.

[19] IBM Combinatorial Test Services, http://www.alphaworks.ibm.com/tech/cts

[20] Hedayat, A.S., N. J. A. Sloane and John Stufken, Orthogonal Arrays: Theory and Applications, Springer Verlag, 1999

[21] G. J. Myers, The Art of Software Testing, Wiley, New York, 1979

[22] Phadke, M.S. "Quality Engineering Using Robust Design", Prentice Hall, Englewood Cliff, NJ. November 1989.

[23] Taguchi, Genichi. "System of Experimental Design", Edited by Don Clausing. New York: UNIPUB/Krass International Publications, Volume 1 & 2, 1987

[24] Software Safety, NASA Technical Standard. NASA-STD-8719.13A, Sept. 1997, http://satc.gsfc.nasa.gov/assure/nss8719_13.html.

[25] Mori, G., Paternò, F., Santoro, C.: CTTE: Support for Developing and Analysing Task Models for Interactive System Design, IEEE Transactions on Software Engineering, Vol. 28, No. 9, IEEE Press, 2002.

[26] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*, Reading, Mass.: Addison-Wesley, 1999.

[27] Hong, H. S. and Lee, I. Automatic test generation from specifications for control-flow and data-flow coverage criteria, in *Proc. Monterey Workshop*, Monterey, Calif.: Naval Postgraduate School (Monterey, Calif., June 2001), pp.230-246.

[28] Wang, Y. and Parnas, D. Simulating the behavior of software modules by trace rewriting**,** *IEEE Trans. Software Eng.* 20, 10 (Oct. 1994), pp. 750-759.

[29] Alfonso, A., Braberman, V., Kicillof, N., and Olivero, A. Visual timed event scenarios, in *Proc. 26th Int. Conf. on Software Engineering*, ACM Press (Edinburgh, Scot., May 2004), pp. 168-177.

[30] McKeeman, W. M. Differential testing for software, *Digital Tech. J.* 10, 1 (1998), pp. 100-107.

[31] Maurer, P. Generating test data with enhanced context-free grammars, *IEEE Software*, July 1990, pp.50-55

[32] S. Fraser and D. Mancl. No silver bullet: Software engineering reloaded. IEEE Software, 25(1):91-94, 2008

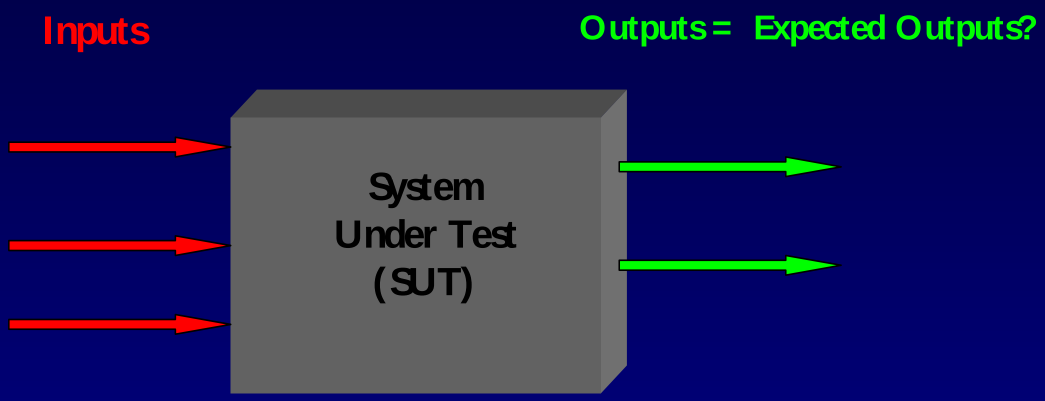
[33] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). "Aspect-Oriented Programming". *Proceedings of the European Conference on Object-Oriented Programming, vol.1241*. pp. 220–242.

New Directions in Software Quality Assurance Automation

Mikhail Auguston

Computer Science Department
Naval Postgraduate School, Monterey, California
maugusto@nps.edu

Black Box Testing



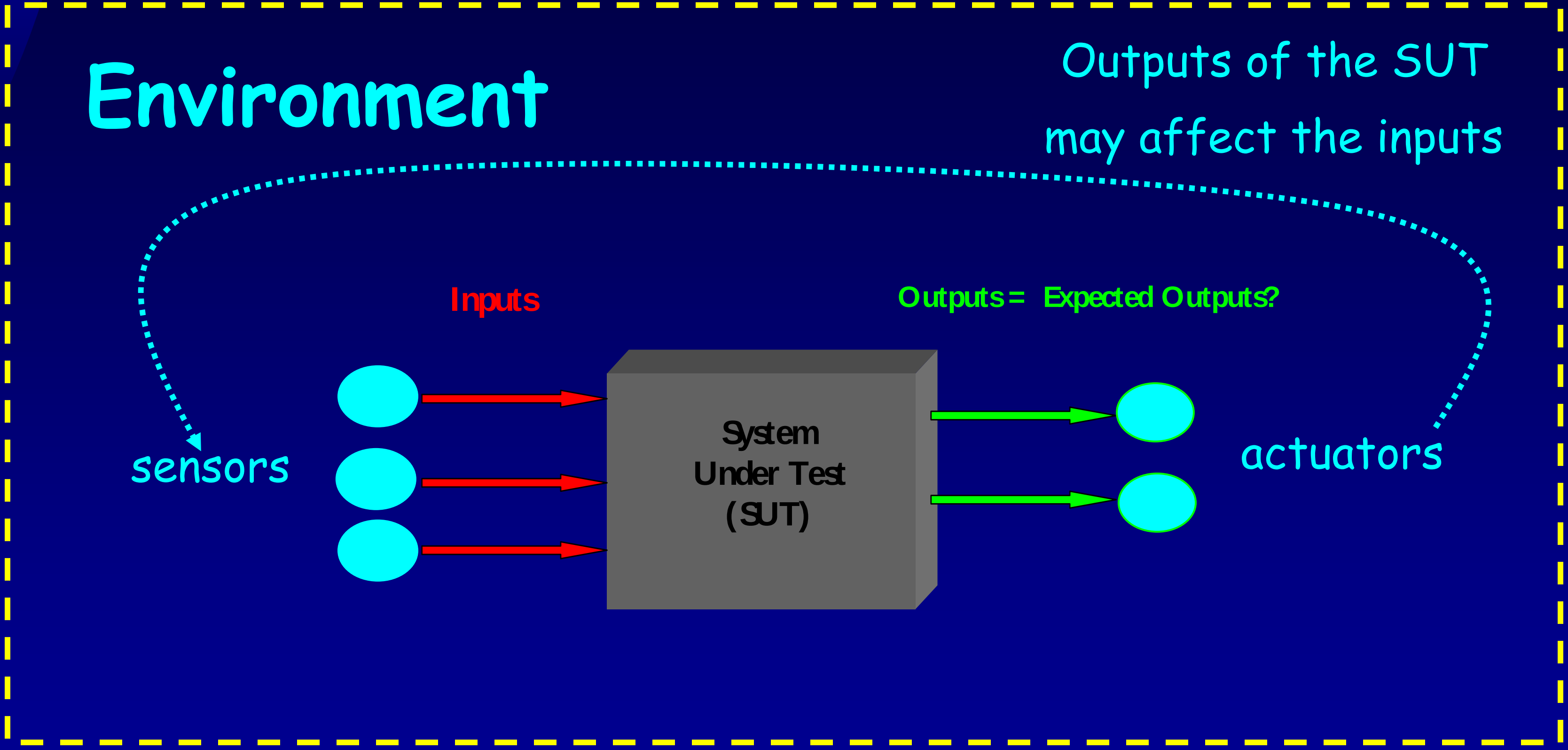
The main problems:

☞How to create test cases

☞How to run a test case

☞How to verify the results of a test run

Black Box testing



The SUT may be a complex reactive real-time C3I system

Testing methodology

- We suggest (pseudo-)random test generation based on the environment models.

- It is best suited for a very special class of programs: reactive and real-time. These programs are of special interest for DoD-related applications.

The model of environment
(an approach to behavior modeling)

An **event** is any detectable action that is executed in the "black box" environment

- An event is a time interval
- An event has attributes: e.g., type, timing attributes, etc.
- There are two basic relations for events:

  precedence and **inclusion**
- The behavior of environment can be represented as a set of events (event trace)

The model of environment

Usually event traces have a certain structure (or constraints) in a given environment

Examples:

1. Shoot_a_gun is a sequence of a Fire event followed by either a Hit or a Miss event
2. Driving_a_car is an event that may be represented as a sequence of zero or more events of types

   go_straight, turn_left, turn_right, or stop

The model of environment

The structure of possible event traces for a given environment can be specified using event grammar

1. Shoot_a_gun::= Fire (Hit | Miss)

   Shooting::= Shoot_a_gun *

2. Driving_a_car::=

   ```
       go_straight
       ( go_straight | turn_left | turn_right ) *
       stop
   go_straight::=  ( accelerate | decelerate | cruise )
   ```

Sequential and parallel events

The precedence relation defines the partial order of events

Two events are not necessary ordered; i.e., they can happen concurrently

Examples

Shoot_a_gun::= Fire (Hit | Miss)

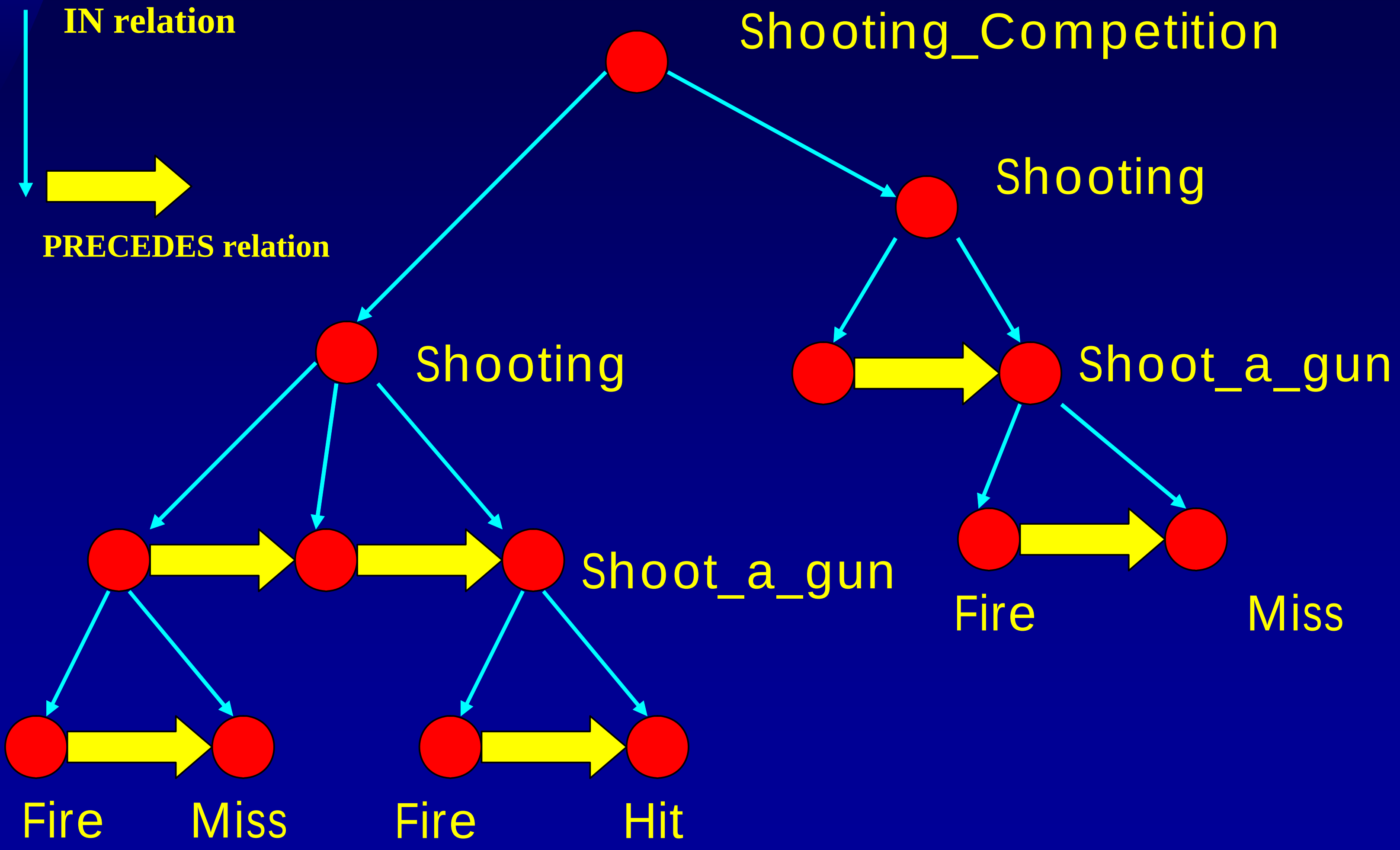
Shooting::= (* Shoot_a_gun *)

Shooting_Competition::= {* Shooting *}

This is a sequence

Those events may be parallel

Visual representation of event trace (not all events and relations are shown...)



Event attributes

```
Shoot_a_gun::=  Fire  (Hit  /Shoot_a_gun. points = Rand[1..10];
                   ENCLOSING Shooting .points += Shoot_a_gun .points; / |
                                   Miss /Shoot_a_gun. points = 0;/)

Shooting::=      / Shooting .points = 0; /
                   (* Shoot_a_gun
                      /Shooting .ammo -=1;/ *) While (Shooting .ammo > 0)

Shooting_Competition ::= /num = 0;/
                   {* /Shooting .id = num++;
                         Shooting .ammo =10;/
                      Shooting *} (Rand[2..100])
```

Production grammars

- ☞ Attribute event grammars (AEG) are intended to be used as a vehicle for automated random event trace generation
- ☞ It is assumed that the AEG is traversed top-down and left-to-right and only once to produce a particular event trace
- ☞ Randomized decisions about what alternative to take and how many times to perform the iteration should be made during the trace generation
- ☞ Attribute values are evaluated during this traversal

Using AEG to generate event traces and inputs to the SUT

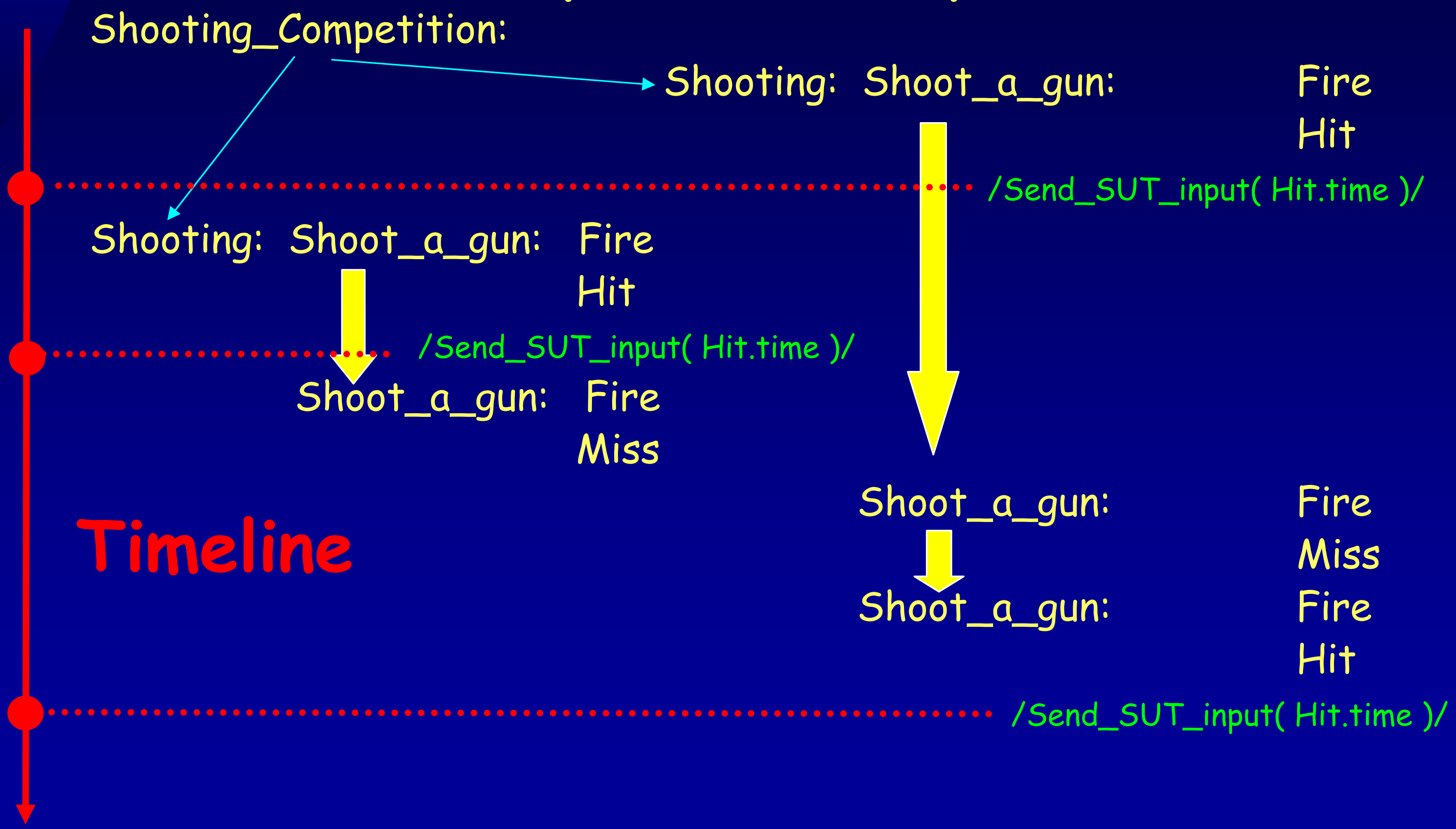
We can provide the probability of selecting an alternative

```
Shoot_a_gun::=  Fire
        ( P(0.3) Hit
        /Send_input_to_SUT( ENCLOSING Shooting .id, Hit .time);/ |
                  -- this simulates SUT sensor input
          P(0.7) Miss )
```

We can generate a large number of event traces satisfying the constraints imposed by the event grammar

Production grammar

The grammar can be used in order to generate event traces and SUT inputs, for example:



Shooting_Competition:

Shooting: Shoot_a_gun: Fire
Hit

/Send_SUT_input(Hit.time)/

Shooting: Shoot_a_gun: Fire
Hit

/Send_SUT_input(Hit.time)/

Shoot_a_gun: Fire
Miss

Shoot_a_gun: Fire
Miss

Shoot_a_gun: Fire
Hit

/Send_SUT_input(Hit.time)/

Use cases

- ☞Event traces are essentially use cases

- ☞Examples of event traces can be useful for requirements engineering, prototyping, and system documentation

Example when SUT outputs are incorporated into the environment model

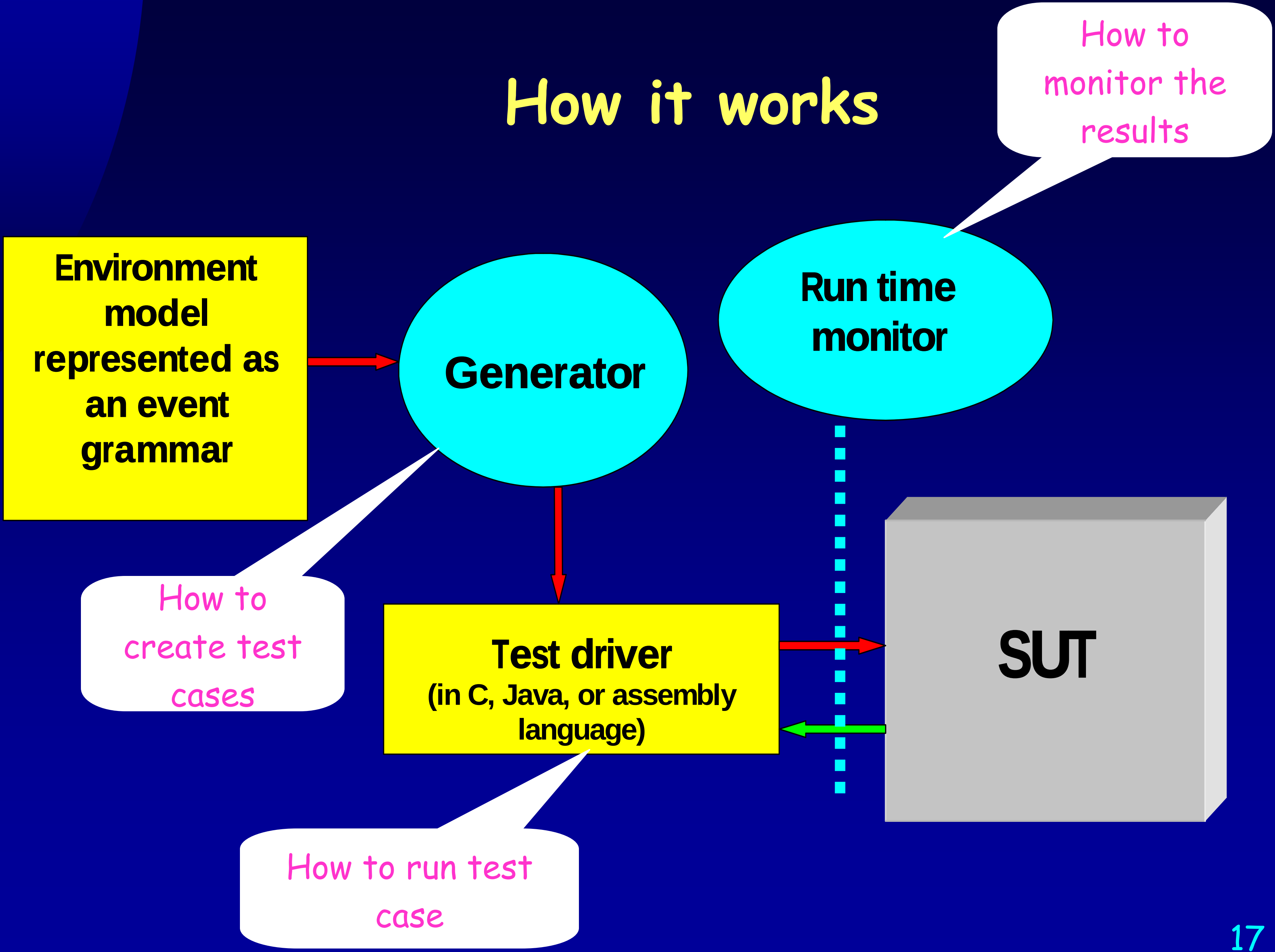
```
Attack::= {* Missile_launch  *} (Rand[1..5])
Missile_launch::= boost   middle_stage  WHEN(middle_stage.completed) Boom
middle_stage::= / middle_stage.completed = true;/
                   (* CATCH    interception_launched (hit_coordinates)
                            -- this external event intercepts SUT output
                      WHEN (hit_coordinates == middle_stage .coordinates )
                      [  P(0.1)   hit_hard
                            / middle_stage.completed= false;
                              send_SUT_input(middle_stage .coordinates);
                                      -- this simulates SUT sensor input
                              Break; / -- breaks the iteration
                      ]
                     OTHERWISE  move
                   *)
move ::= /adjust (ENCLOSING middle_stage .coordinates) ;
          send_SUT_input( ENCLOSING middle_stage .coordinates);
           -- this simulates SUT sensor input
          DELAY(50 msec); /
```

Prototype implementation

The test generator based on attributed event grammars has been implemented at NPS

It takes an AEG and generates a test driver in Java.

How it works


How to monitor the results
Environment model represented as an event grammar
Generator
Run time monitor
How to create test cases
Test driver
(in C, Java, or assembly language)
SUT
How to run test case

Software safety assessment

- ☞ In the previous example, the Boom event will occur in certain scenarios depending on the SUT outputs received by the test driver and random choices determined by the given probabilities

- ☞ If we run large enough number of (automatically generated) tests, the statistics gathered gives some approximation for the risk of getting to the hazardous state. This becomes a very constructive process of performing experiments with SUT behavior within the given environment model ("software-in-the-loop" simulations)

Qualitative Risk Analysis

```
Attack::= { Missile_launch } * (<=N)
Missile_launch::= boost   middle_stage  Boom
middle_stage::= ( CATCH  interception_launched(hit_coordinates)
                   -- this external event intercepts SUT output
                   [  P(p1) hit_hard
                     /send_hit_input(middle_stage.coordinates);
                      Break; / ]
                   OTHERWISE move
                  )*
```

- ☞ **Experimenting with increasing or decreasing N and p1 we can conclude what impact those parameters have on the probability of a hazardous outcome, and find thresholds for SUT behavior in terms of N and p1 values**

Qualitative Risk Analysis (2)

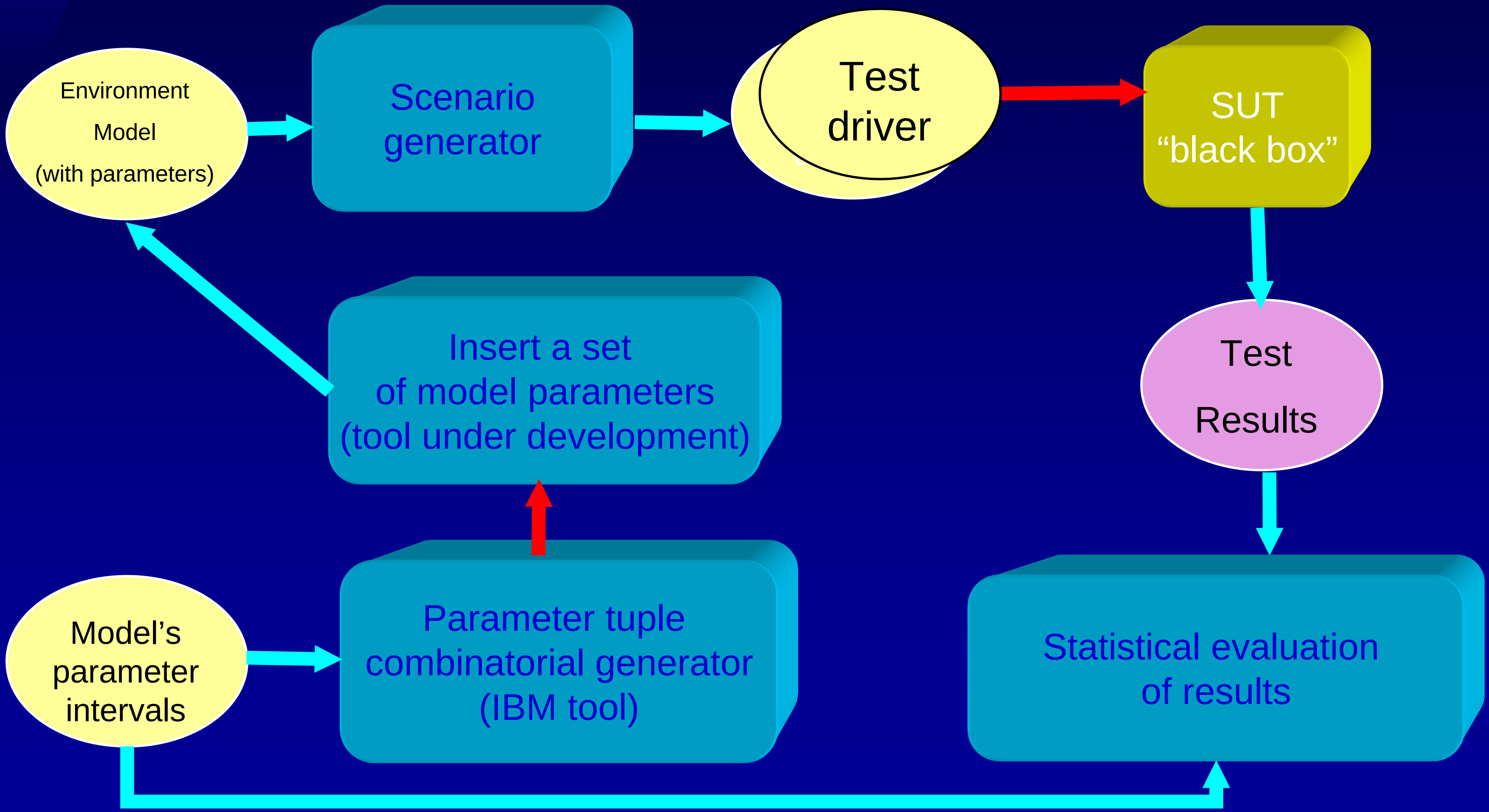
- ☞ We can change some parameters in the model and repeat the set of tests. If the frequency of reaching a hazardous state changes, we can find out how the parameter values influence the probability to reach a hazard state

- ☞ We suggest to use the **combinatorial testing technique** based on orthogonal arrays, an approach well familiar to statisticians

Qualitative Risk Analysis (3)

- The same conjecture that stipulates that the fault in behavior of the SUT in most cases depends either on a **single parameter value** or on an interaction of a **pair of parameter values** could be applied to the system safety testing. This conjecture still has to be verified by experiments
- Combinatorial approach will significantly **reduce the number** of experiments needed to establish statistically sound conclusions about probabilities to reach hazard state for different environment model settings
- In order to apply combinatorial testing techniques the values of model parameters have to be split into a **finite number of equivalence classes**, a technique well known in software component testing

SUT safety assessment with automated scenario generation


Environment
Model
(with parameters)
Scenario generator
Test driver
SUT
“black box”
Insert a set of model parameters (tool under development)
Test
Results
Model’s parameter intervals
Parameter tuple combinatorial generator (IBM tool)
Statistical evaluation of results

The main advantages

- ☞ The whole testing process can be automated
- ☞ The AEG formalism provides powerful high-level abstractions for environment modeling
- ☞ It is possible to run many more test cases with better chances to succeed in exposing an error
- ☞ It addresses the regression testing problem – generated test drivers can be saved and reused.
- ☞ AEG is well structured, hierarchical, and scalable
- ☞ The environment model itself is an asset and could be reused

Why it will fly

- ☞ Environment model specified by AEG provides for high-level domain-specific formalism for testing automation
- ☞ The generated test driver is efficient and could be used for real-time test cases
- ☞ Different environment models can be designed; e.g., for testing extreme scenarios by increasing probabilities of certain events, or for load testing
- ☞ Experiments running SUT with the environment model provide a constructive method for quantitative and even qualitative software safety assessment
- ☞ Environment models can be designed on early stages of system design, can provide environment simulation scenarios or use cases, and can be used for tuning the requirements and for prototyping efforts

Questions, please?

Backup slides

Example – simple calculator environment model

```
Use_calculator:   (* Perform_calculation *);
Perform_calculation:
        Enter_number  Enter_operator  Enter_number
        WHEN (Enter_operator.operation == '+')
        / Perform_calculation.result =
                Enter_number[1].value + Enter_number[2].value; /
        ELSE
        / Perform_calculation.result =
                Enter_number[1].value - Enter_number[2].value; /
        [ P(0.7) Show_result ];
```

Example – simple calculator environment model

```
Enter_number:           / Enter_number.value= 0; /
        (* Press_digit_button
          / Enter_number.digit = RAND[0..9];
            Enter_number.value =
               Enter_number.value * 10 + Enter_number.digit;
           enter_digit(Enter_number.digit); /  *) Rand[1..6];
Enter_operator:
                 ( P(0.5) / enter_operation('+');
                            Enter_operator .operation= '+'; / |
                   P(0.5) / enter_operation('-');
                            Enter_operator .operation= '-'; / ) ;

Show_result:    /show_result();/ ;
```

Example 2 –Infusion Pump model

```
CARA_environment:   { Patient, LSTAT, Pump };

Patient:            / Patient.bleeding_rate= BR; /
                    (*  / Patient.volume +=
                              ENCLOSING CARA_environment ->
                                     Pump.Flow – Patient.bleeding_rate;
                           Patient.blood_pressure =
                                     Patient.volume/50 – 10;
                           Patient.bleeding_rate += RAND[-9..9]; /
                         WHEN (Patient.blood_pressure > MINBP)
                              Normal_condition
                         ELSE
                              Critical_condition
                    *) [EVERY 1 sec] ;
```

Example 2 –Infusion Pump model

```
LSTAT:          Power_on / send_power_on(); /
                (* / send_arterial_blood_pressure(
                         ENCLOSING CARA_environment->
                                  Patient.blood_pressure); /
                *) [EVERY 1 sec] ;

Pump:           Plugged_in
                /  send_plugged_in();
                   Pump.rotation_rate = RR;
                   Pump.voltage = V; /
                { Voltage_monitoring, Pumping };
```

Example 2 –Infusion Pump model

```
Voltage_monitoring:        (*  / ENCLOSING Pump.EMF_voltage =
                                       ENCLOSING Pump.rotation_rate * REMF;
                                  send_pump_EMF_voltage(
                                       ENCLOSING Pump.EMF_voltage); /
                           *) [ EVERY 5 sec]  ;
Pumping:
                           (* / ENCLOSING Pump. rotation_rate =
                                       ENCLOSING  Pump. voltage * VRR;
                                 ENCLOSING Pump. flow =
                                       ENCLOSING Pump. rotation_rate * RRF; /
                             CATCH set_pump_voltage( ENCLOSING Pump.voltage)
                             Voltage_changed
                             [ P(p1)  Occlusion
                                       / ENCLOSING Pump.occlusion_on = True;
                                         send_occlusion_on(); / ]
                             WHEN ( ENCLOSING Pump.occlusion_on)
                             [ P(p2) / ENCLOSING Pump.occlusion_on  =False;
                                        send_occlusion_off(); / ]
                           *) [EVERY 1 sec] ;
```

Backup slides
Program monitoring and test oracles

(How to verify the results of a test run)

Objective: to develop unifying principles for program monitoring activities

Suggested solution: to define a precise model of program behavior as a set of events – event trace

Monitoring activities in software design can be implemented as computations over program execution traces.

Examples:

- Assertion checking (test oracles)
- Debugging queries
- Profiles
- Performance measurements
- Behavior visualization

Program Behavior Models

- ☞ Program monitoring activities can be specified in a uniform way using program **behavior models** based on the event notion
- ☞ An **event** corresponds to any detectable action; e.g., subroutine call, expression evaluation, message passing, etc. An event corresponds to a time interval
- ☞ Two partial order binary relations are defined for events: **precedence** and **inclusion**
- ☞ An event has **attributes**: type, duration, program state at beginning or end of the event, value,…

Program Behavior Models

- Event grammar specifies the constraints on configurations of events generated at the run time (in the form of axioms, or “lightweight semantics” of the target language)

- Some axioms are generic; e.g., transitivity and distributivity

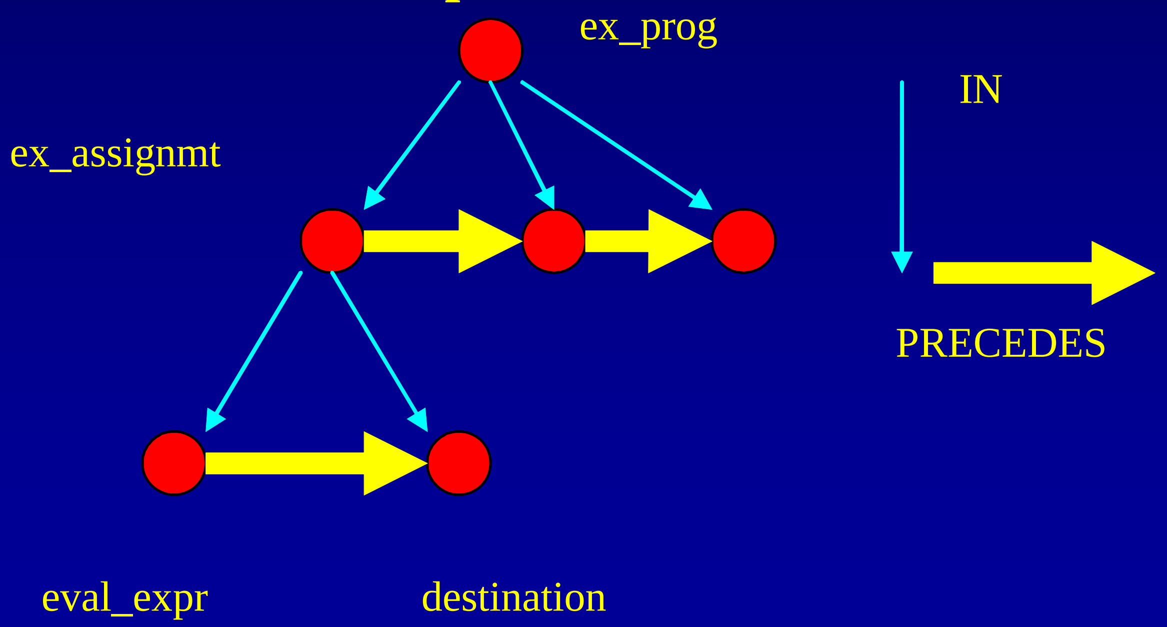
A PRECEDES B and B PRECEDES C → A PRECEDES C

A IN B and B PRECEDES C → A PRECEDES C

Example of an Event Grammar

```
ex_prog:: ex_stmt *
ex_stmt:: ex_assignmt | ex_read_stmt | …
ex_assignmt:: eval_expr  destination
```

Example of an event trace



Program Monitoring

- Monitoring activities: assertion checking, profiles, performance measurements, dynamic QoS metrics, visualization, debugging queries, intrusion detection
- Program monitoring can be specified in terms of computations over event traces
- We introduce a specific language FORMAN to describe computations over event traces (based on event patterns and aggregate operations over events)

FORMAN language

- Event patterns

```
x: func_call & x.name == “A”
eval_expr :: ( variable )
```

- List of events

```
[ exec_assignmt FROM ex_prog]
```

- List of values

```
[ x: exec_assignmt FROM ex_prog APPLY x.value]
```

FORMAN language

- Aggregate Operations

```
MAX/[ x: exec_assignmt FROM ex_prog APPLY x.value]
```

```
AND/[ x: exec_assignmt FROM ex_prog APPLY x.value > 17]
```

Or

```
FOREACH x: exec_assignmt FROM ex_prog x.value > 17
```

Examples

1) Profile

```
SAY( "Number of function A calls is "
   CARD[ x: func_call & x.name == "A"
                        FROM ex_prog ]
```

Event pattern

Aggregate operation

2) Generic debugging rule (typical error description)

```
FOREACH e: eval_expr :: (v: variable)
                              FROM ex_prog
  EXISTS d: destination FROM e.PREV_PATH
            v.source_code = d.source_code
   ONFAIL SAY("Uninitialized variable "
         v.source_code "is used in expression " e)
```

Event attribute

Examples

3) Debugging query

```
SAY("The history of variable x "
[d: destination & d.source_code == "x" FROM ex_prog
  APPLY d.value ] )
```

4) Traditional debugging print statements

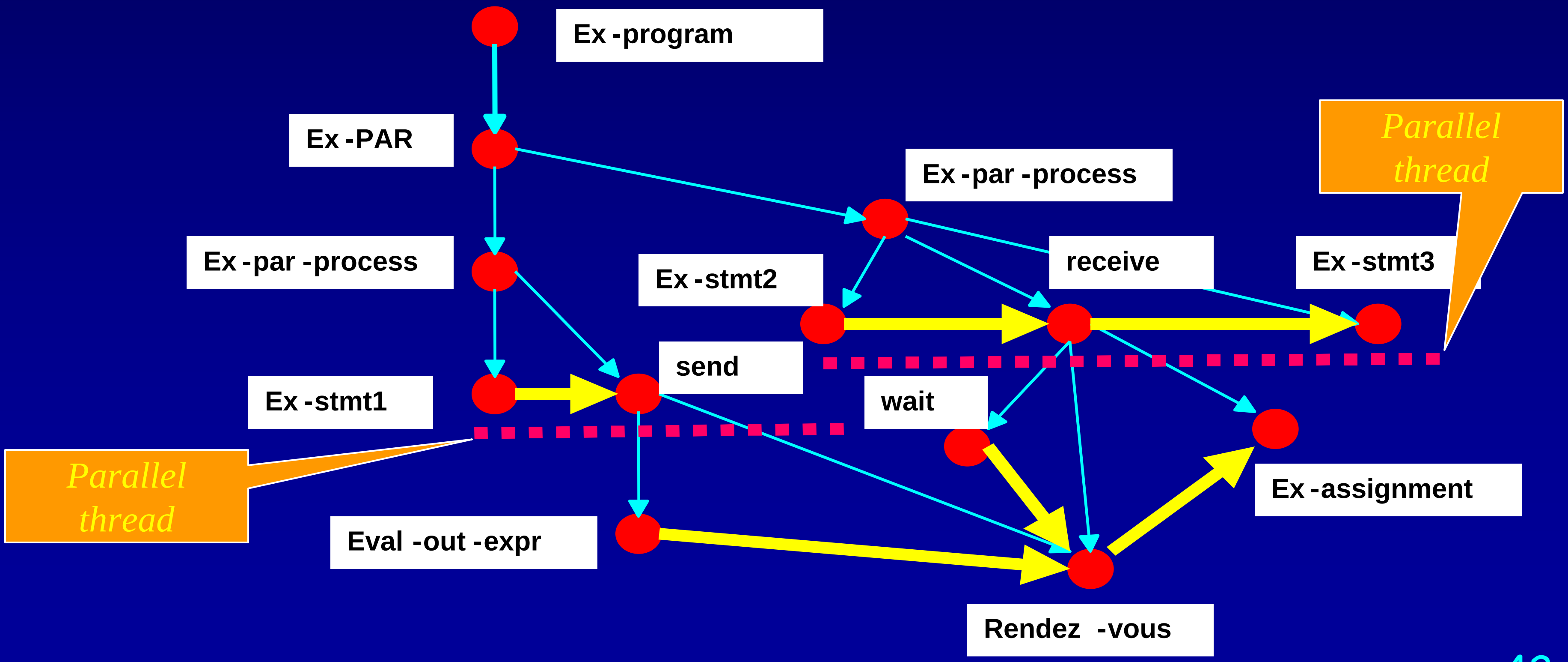
```
FOREACH f: func_call & f.name == "A"
                                  FROM ex_prog
  f.value_at_begin(
             printf("variable x is %d\n", x) )
```

Event attribute

Expression
Evaluated at the run time

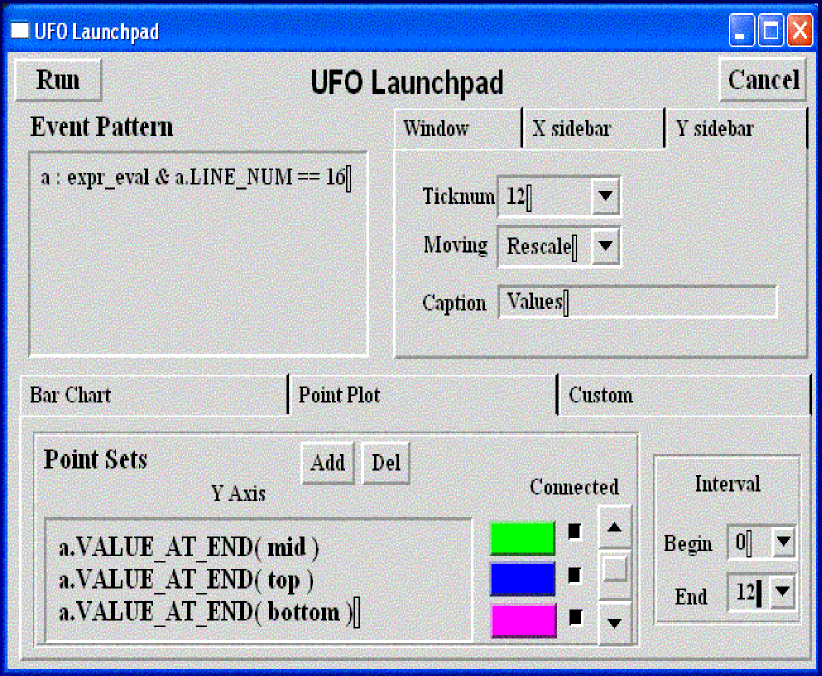
Example of event trace representing a synchronization event (send/receive a message)

```
par     --launches two parallel processes
        seq     -- first parallel thread
                stmt1
                channel1 ! Out-expr    -- sends a message
                …
        seq     -- another parallel thread
                stmt2
                channel1 ? Var         -- receives a message
                …
```



Program visualization (UFO project)

Visualization prototype for Unicon/ALAMO (Jointly with C.Jeffery, NMSU)



UFO Launchpad

Event Pattern

a : expr_eval & a.LINE_NUM == 16

Ticknum 12

Moving Rescale

Caption Values

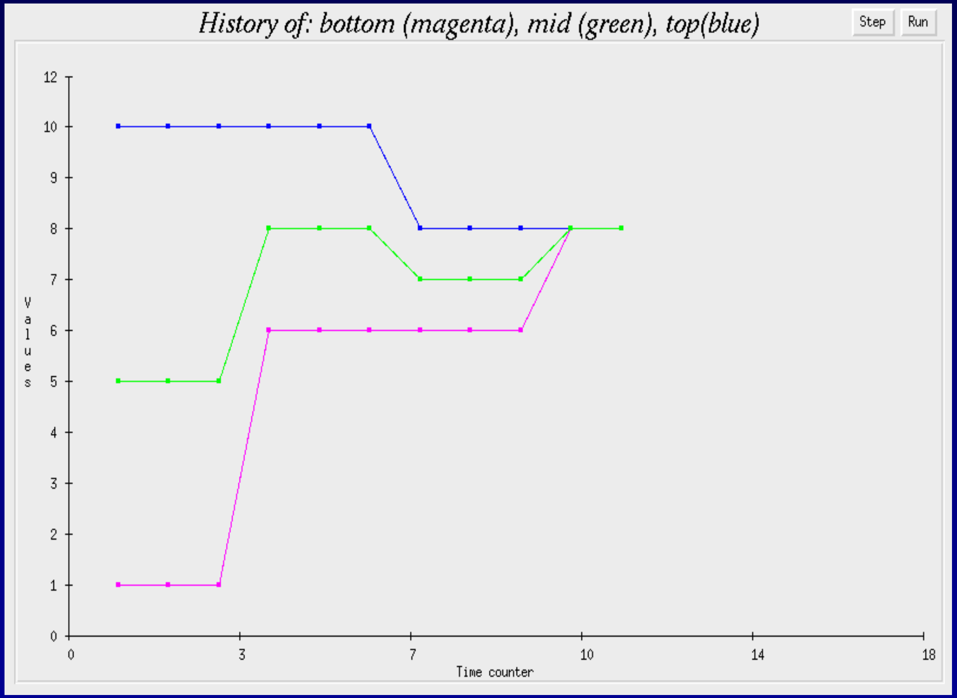
Point Sets

Y Axis	Connected
a.VALUE_AT_END(mid)	🟩 ■
a.VALUE_AT_END(top)	🟦 ■
a.VALUE_AT_END(bottom)	🟪 ■

Interval

Begin 0

End 12



Point plot example for a binary search program

The novelty claims of our approach

- **Uniform framework** for program monitoring based on precise behavior models and event trace computations
- Computations on the event traces can be implemented in a **nondestructive** way via automatic instrumentation of the source code or even of the executables (Dyninst approach)
- Can specify **generic trace computations**: typical bug detection, dynamic QoS metrics, profiles, visualization, …
- Both **functional** and **non-functional** requirements can be monitored
- Yet another approach to the **aspect-oriented** paradigm

Accomplished projects and work in progress

- Assertion checker for a Pascal subset (via interpreter)
- Assertion checker for the C language (via source code instrumentation)
- Assertion checker and visualization tool for the Unicon language (via Virtual Machine monitors)
- Dynamic QoS metrics, UniFrame project (via glue and wrapper instrumentation), funded by ONR
- Intrusion detection and countermeasures (via Linux kernel library instrumentation using NAI GSWTK), funded by the Department of Justice Homeland Security Program
- Automated test driver generator for reactive real time systems based on AEG environment models, funded by Missile Defense Agency

Some publications

- M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, 2nd Int’l Workshop on Automated and Algorithmic Debugging, AADEBUG'95, Saint-Malo, May 1995, pp. 277-291.

- M. Auguston, A. Gates, M. Lujan, Defining a Program Behavior Model for Dynamic Analyzers, 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, June 1997, pp. 257-262.

- M.Auguston, Assertion Checker for the C Programming Language based on computations over event traces, in Proceedings of the Fourth International Workshop on Algorithmic and Automatic Debugging, AADEBUG'2000, Munich, August 28-30, 2000, pp.90-99 on-line proceedings at http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html

- M. Auguston, C. Jeffery and S. Underwood. A Framework for Automatic Debugging. Proceedings of the IEEE 17th International Conference on Automated Software Engineering, ASE'02, Edinburgh, September 2002, IEEE Computer Society Press, pp.217-222.

- Mikhail Auguston, James Bret Michael, Man-Tak Shing, Environment Behavior Models for Scenario Generation and Testing Automation, in Proceedings of the First International Workshop on Advances in Model-Based Software Testing (A-MOST'05), the 27th International Conference on Software Engineering ICSE’05, May 15-16, 2005, St. Louis, USA, http://a-most.argreenhouse.com, also in the ACM Digital Library

Summary of the event grammar approach

- ☞ Behavior models based on event grammars provide a uniform framework for software testing and debugging automation
- ☞ Can be implemented in a nondestructive way via automatic instrumentation
- ☞ Automated tools can be built to support all phases of the testing process
- ☞ Provides a good potential for reuse: environment models, generic debugging rules, test drivers for regression testing
- ☞ Provides high-level abstractions for testing and debugging tasks, hence is easy to learn and use
- ☞ Well suited for reactive real-time system testing

Why bother?

Testing and debugging consume more than 50% of total software development cost.

If the proposed research is transferred into practice and reduces costs by 1% of the 50% of the $400 billion software industry, the potential economic impact would be around $2 billion per year.